Basanta-Val, P., García-Valls, M., Simple Multiplexing Headers for the JRMP Stream Subprotocol". IEEE Latin America Transactions, (2016), 14(2), 1005-1010.
DOI: http://dx.doi/10.1109/TLA.2016.7437251

Simple Multiplexing Headers for the JRMP Stream Subprotocol

P. Basanta and M. García

Abstract— **This article deals with a simple optimization for a level-5 protocol called JRMP (Java's Remote Method Protocol), which is used in a distribution model named Java's RMI (Java's Remote Method Invocation). The main JRMP subprotocol, namely Stream, has been enhanced with a simple and direct multiplexing mechanism that offers the possibility of transferring several parallel request-response interactions without opening new TCP/IP connections. The overhead required to process headers and the advantages stemmed from the approach in terms of response-time are explored on a switched-ethernet benchmark application.**

Keywords— **Protocol implementation, empirical evaluation, JRMP, real-time Java.**

I. INTRODUCCIÓN

EXISTE en los sistemas de tiempo real ([1][2, 3][4-6]) una tendencia cada vez más marcada hacia la utilización de infraestructuras de comunicación que requieren un comportamiento que conjugue predictibilidad con flexibilidad. En un principio reinó la predictibilidad mediante mecanismos de reservas en las comunicaciones, en fases iniciales de la comunicación que eran reutilizados por cada comunicación inter nodal. Sin embargo, en la actualidad, la necesidad de hacer sistemas más flexibles que interactúen con Internet ([7-9]) hace que se añadan progresivamente soluciones que permiten ofrecer un mayor rendimiento de la infraestructura subyacente.

En este campo, este trabajo se alinea con los protocolos que dan soporte a los middlewares de tiempo real de próxima generación [1]. En especial se centra en Java distribuido de tiempo real ([10][11]) que intenta proveer comunicación entre diferentes nodos equipados con máquinas virtuales Java de tiempo real. Este middleware utiliza un protocolo de comunicaciones llamado JRMP (Java's Remote Method Protocol) [12][13, 14] que comunica a nivel 5 diferentes nodos Java.

Desde el punto de vista de las aplicaciones de tiempo real, dicho soporte puede ser insuficiente al no existir un mapeo claro entre conexiones TCP/IP y las tareas con requisitos de tiempo real. Diversos investigadores han abordado el tema ([15][16]) propagando la urgencia, en forma de prioridad, de sus peticiones al servidor mediante la modificación del protocolo JRMP. Otros investigadores han propuesto que se recuperen otros protocolos ya existentes ([17][18][19]) dentro de RMI. La idea sobre las que trabajan es que hay diferentes subprotocolos, algunos ya existentes en JRMP (SingleOp y Multiplex), y que se añadan otros nuevos (e.g. *ConnectionLess* [18][19]) que ofrezcan una multiplexación eficiente en las comunicaciones. Como consecuencia de estos nuevos protocolos la aplicación tendrá que seleccionar el que más le interese como parte de su configuración.

En este artículo se explora la utilización de un protocolo similar a *ConnectionLess* como sustituto del protocolo Stream. Experiencias de comparación previa ([18] [19]) sugieren que el rendimiento de *ConnectionLess* debería de estar cerca del comportamiento de *Stream*. Sin embargo, no se ha cuantificado en términos de ganancia o de pérdidas los costes que tendría realizar dicha sustitución. Este es el objetivo de este artículo: el de cuantificar el coste del mecanismo de multiplexacion eficientemente cuando ejecuta directamente sobre el principal protocolo de JRMP: *Stream*. Hasta ahora los trabajos realizados con *ConnectionLess* se han centrado más en comparar el coste en aquellos casos en el que el protocolo evita que se negocien nuevas conexiones TCP/IP, en los cuales ha mostrado un gran rendimiento. Por tanto la exploración realizada en este artículo complementa el trabajo previo.

Este resultado tiene un impacto en el trabajo que se está realizando en diferentes infraestructuras para Java de tiempo real distribuido ([17,20][21][15]). Actualmente, estas aproximaciones utilizan el protocolo JRMP con *Stream* (con algunas cabeceras adicionales para tiempo real). Si los costes computacionales extras introducidos por las cabeceras de multiplexación propuestos son competitivos, se podría incorporar directamente sobre las comunicaciones ya existentes, sin necesidad de nuevos subprotocolos en JRMP.

El resto de este artículo se enfoca en presentar una estrategia sencilla de cabeceras para el subprotocolo de JRMP denominado *Stream*. La Sección II introduce el contexto de la evaluación dentro de la arquitectura por capas de un middleware para Java de tiempo real. La Sección III analiza el problema y propone un esquema sencillo de cabecera de multiplexación. Tras ello, se evalúa la propuesta en un escenario de trabajo descrito en la Sección IV. La Sección V explora aquellos trabajos más relacionados con el propuesto. Por último, la Sección VI resume los principales logros realizados en el trabajo y propone líneas futuras de actuación a considerar a corto plazo.

II. CONTEXTO DE LA EVALUCION

A. Java de Tiempo Real Distribuido

El contexto de este protocolo gira alrededor de Java de tiempo real. Actualmente, aunque existe un intento de especificación denominado DRTSJ [22] (The Distributed Real-Time

Specification for Java), de forma práctica no existen implementaciones públicas que le den un soporte adecuado. Aún así existen algunas implementaciones ([15][23]) que ofrecen algunos resultados sobre el rendimiento que ciertos bloques funcionales de DRTSJ pueden ofrecer.

Este trabajo se alinea con el middleware denominado DREQUIEMI [20][24]. DREQUIEMI ofrece control sobre los recursos (memoria, procesador y red) utilizados durante una comunicación remota. DREQUIEMI está basado en un modelo capas, simplificado para que se ajuste mejor a Java RMI (Remote Method Invocation).

DREQUIEMI (Fig. 1) asume la existencia de una capa para Java de tiempo real basada en RTSJ (The Real-Time Specification for Java) que se extiende con un soporte para acceso a la red. El acceso a cada uno de estos elementos puede ser modulado por la capa de Java de tiempo real que permite la gestión de recursos localmente. DREQUIEMI extiende esa gestión a gestores de Java de tiempo real que permiten la gestión de memoria, procesador y red mediante estrategias basadas en almacenes (denominadas *pools*) y que permiten la reutilización de dichos recursos. Estos recursos son utilizados por los servicios de DREQUIEMI (invocación remota, recolector de basura, nombramiento y eventos distribuidos) para ofrecer un funcionamiento predecible. Por último, la capa de aplicación está basada en componentes que facilitan la utilización de componentes implementados por terceros en una aplicación.

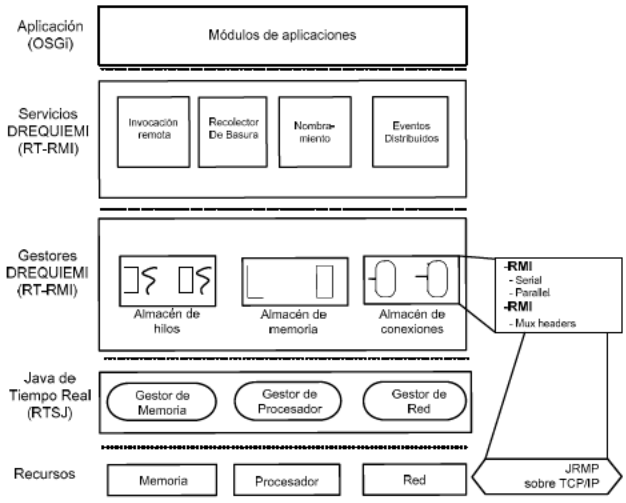

Figura 1. Comunicaciones dentro de la arquitectura DREQUIEMI

Dentro de toda esta arquitectura, el presente artículo se centra en la red y más concretamente en el protocolo sobre el que se asientan las comunicaciones en RMI. En la actualidad, las comunicaciones en JRMP se hacen a través de un módulo que permite empaquetar invocaciones remotas en mensajes tipo petición-respuesta que corren sobre diferentes subprotocolos, que a su vez se asientan sobre conexiones TCP/IP. Aunque en la especificación de JRMP reconoce la existencia de tres posibles protocolos (a saber: *SingleOp*, *Stream*, y *Multiplex*), de facto las implementaciones de Java RMI estándar sólo soportan *Stream*.

III. PROBLEMA Y APROXIMACIÓN

A. *Mapeo entre conexiones TCP/IP y comunicaciones JRMP*

Stream permite la reutilización de una conexión TCP/IP de tal manera que tras una invocación remota (es decir un par `CALL-RETURN`) puede ser reutilizada para realizar otra invocación desde el mismo cliente. Sin embargo, no es posible que varias peticiones `CALL-RETURN` provenientes de un cliente se entremezclen. Si ese es el objetivo, una posible solución es tener una conexión dedicada a cada tipo de comunicación que se establece desde el cliente.

La Fig. 2 ejemplifica la relación existente entre el protocolo JRMP y la conexión TCP/IP subyacente. La figura muestra todos los pasos necesarios para enviar un mensaje de invocación remota en JRMP. En primer lugar hay que negociar mediante mensajes `SYN`, `SYN-ACK` y `ACK` la conexión TCP (usando un protocolo ida vuelta ida). Una vez establecida la conexión se puede enviar mediante mensajes JRMP `CALL` y `RETURN` la petición y la respuesta de la invocación. Por último, se debe de cerrar la conexión TCP mediante tres mensajes tipo FIN, FIN-ACK y ACK.

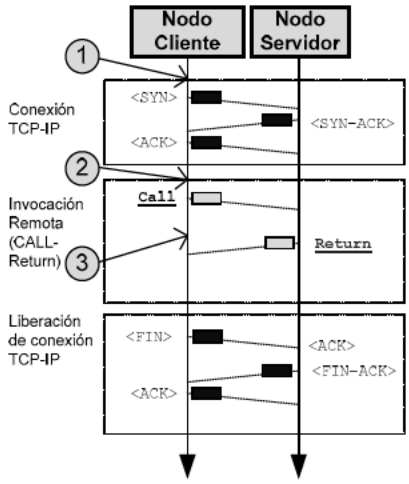

Figura 2. Estrategias de mapeo entre mensajes RMI y TCP/IP

Desde el punto de que varios hilos situados en un mismo cliente puedan invocar concurrentemente a un servidor realizando invocaciones, en JRMP existen tres formas naturales (marcadas como 1 2 y 3 en la Fig. 2) de ofrecer dicho soporte:

1. Se puede negociar una conexión TCP/IP por conexión. Eso hace que se abra una conexión por cada invocación, lo que hace que las latencias aumenten y el rendimiento baje notablemente.
2. Se puede reutilizar una conexión que ya esté abierta y que no esté siendo utilizada en ningún proceso de invocación remota. Esto obliga a que exista un almacén de conexiones TCP/IP que son reutilizadas para realizar cada invocación remota.
3. Por último, se puede dotar a cada mensaje de una cabecera de multiplexación que permita que varias invocaciones a la misma máquina coexistan en un determinado nodo. Dicha cabecera de multiplexación

conseguiría utilizar una única conexión TCP/IP para enviar todos los datos entre cada par cliente-servidor.

Los diferentes subprotocolos JRMP ofrecen diferentes formas de soportar dichos patrones de paralelismo:

1. Una opción es utilizar una variante del subprotocolo JRMP conocida como *SingleOp*. Actualmente, el subprotocolo *SingleOp* ha sido descartado por ser altamente ineficiente.
2. Otra opción es la ofrecida por defecto en Java RMI y se denomina *Stream*. *Stream* permite la reutilización de conexiones y desde el punto de vista del tiempo real obliga a que se establezca un número máximo de conexiones entre cliente y servidor.
3. Por último, la tercera opción también está parcialmente contemplada por JRMP y se denomina el subprotocolo *Multiplex*. El problema de este subprotocolo es que emula un protocolo de comunicación orientado a conexión sobre sobre TCP/IP lo que acaba redundando en una mayor sobrecarga. En la actualidad, esta opción no es utilizada en las implementaciones de RMI.

Por último, en la literatura aparece el subprotocolo *ConnectionLess* ([18][19]) como una forma de multiplexación eficiente que permite la invocación paralela desde diferentes clientes. Este protocolo aparece descrito como una alternativa a los tres protocolos anteriores de tal manera que el cliente podría escoger entre la utilización de Stream o de *ConnectionLess* dependiendo del tipo de aplicación que esté desarrollando.

Los buenos resultados obtenidos cuando este *ConnectionLess* se compara con el resto de subprotocolos han llevado a estos investigadores a considerar el coste que tendría la modificación de *Stream* para que integrase de forma nativa las cabeceras de multiplexación ofertadas por *ConnectionLess*.

B. Integración Directa de Cabeceras de Multiplexación en el subprotocolo Stream

De esta manera, a nivel trama, el cambio principal que se haría sería el de añadir la cabecera de multiplexación directamente sobre los mensajes utilizados para la invocación remota (Fig. 3). En dicha invocación remota, los pares de la invocación serían iguales, permitiendo intercalar invocaciones remotas procedentes de un mismo nodo cliente.

La principal limitación de la propuesta sucede cuando los mensajes transmitidos son muy grandes. En este caso la estrategia de multiplexación puede provocar inversiones de prioridad. Este problema es compartido por las anteriores versiones del protocolo, que necesitan procesar las cabeceras para poder continuar.

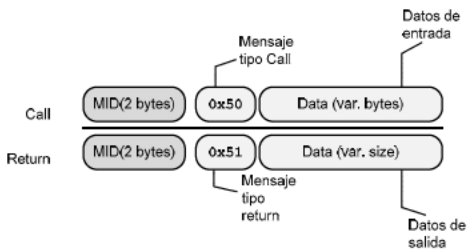

Figura 3. Técnica de Identificadores de Multiplexación aplicada sobre Mensajes tipo CALL y RETURN de JRMP

La Fig. 3 ejemplifica también las fuentes de sobrecarga computacional que experimentarán las comunicaciones multiplexadas. La introducción de este mecanismo acarrea una sobrecarga debida a las propias cabeceras de multiplexación. Estas cabeceras han de ser procesadas propiamente por el cliente y el servidor y suponen un sobrecoste constante en las comunicaciones. Ese sobrecoste junto al beneficio temporal se analiza de forma explícita en la evaluación empírica de de esta sección.

IV. IMPLEMENTACIÓN Y EVALUACIÓN

Se ha alternado la implementación existente para Java RMI y se la ha dotado de cabeceras de multiplexación como las descritas en la Sección II. De tal manera que se han generado dos versiones del software, una que permite la utilización de *Stream* y otra que permite la ejecución de invocaciones en paralelo y que se denominada *mux-Stream*.

El escenario estudiado (Fig. 4) se compone de un sistema operativo de tiempo real, sobre éste corre una máquina virtual para Java de tiempo real y comunicaciones tipo RMI. Dichas comunicaciones han sido modificadas para que se integren opcionalmente las cabeceras multiplexoras descritas en las secciones previas. Sobre dichas comunicaciones se montan aplicaciones distribuidas encargadas de enviar tráfico

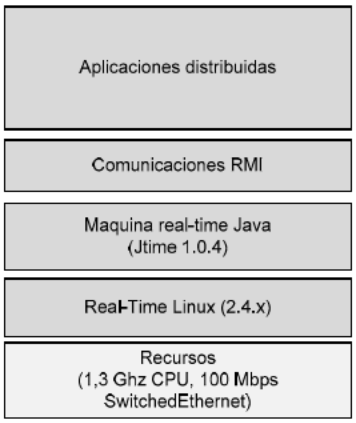

Figura 4. Pila software de referencia utilizada durante la evaluación

El objetivo principal de esta evaluación es estudiar la relación coste-beneficio introducido por la mux-Stream de forma empírica:

- El beneficio se mide como la reducción en tiempo que se obtiene por no tener que negociar de forma dinámica otra conexión TCP/IP para un flujo Stream. Cuanto más alto sea este parámetro, más justificado estará que se introduzca este tipo de soporte.
- El coste se mide como el tiempo extra que es necesario para generar, recuperar y procesar las cabeceras de multiplexación dentro de una invocación remota. Idealmente, cuanto más bajo sea este tiempo, mejor será el rendimiento del protocolo diseñado.

A. *Evaluación de la relación Coste-Beneficio*

La evaluación de esta relación coste-beneficio comienza viendo donde se producen los mayores costes y beneficios. Eso sucede cuando el protocolo tiene una carga mínima de datos de aplicación (cuando la aplicación envía una invocación remota a un objeto remoto que no recibe ni envía ningún dato). En este caso (ver Tabla I y después la Fig. 5), las cabeceras de multiplexación introducen un sobrecoste cercano a los 112 µs que representa un 21% sobre el coste de la utilización del subprotocolo stream. El beneficio también alcanza en este punto el máximo porcentual donde el tiempo ahorrado puede llegar a 1097 µs, lo que supone un ahorro de algo más de dos veces el tiempo de respuesta. Al aumentar la carga de la aplicación (que envía paquetes mayores) tanto el peso de los costes como sus beneficios se van diluyendo (ver Fig. 5).

TABLA I COSTE Y BENEFICIO MÁXIMOS DEL PROTOCOLO EN UNA RED 100-MBITS SWITCHED-ETHERNET CON MAQUINAS (2X) 1 3 GHZ PENTIUM CON RTLINUX Y JTIME

	Coste en µs	Coste/Beneficio sobre Stream (%)
Stream	520 µs	-
Stream + Cabeceras (MID)	632 µs	112 µs → +21%
Stream (conexión TCP/IP)	1617 µs	1097µs → +211%

Un punto muy importante es el punto donde los paquetes TCP/IP sufren fragmentación (esto se corresponde con paquetes IP de 680 bytes en la infraestructura propuesta). En este primer tramo se concentran las invocaciones remotas que se pueden soportar con dos paquetes IP. En este punto el coste constante introducido es del 10% extra y el beneficio supera ligeramente el 80 %.

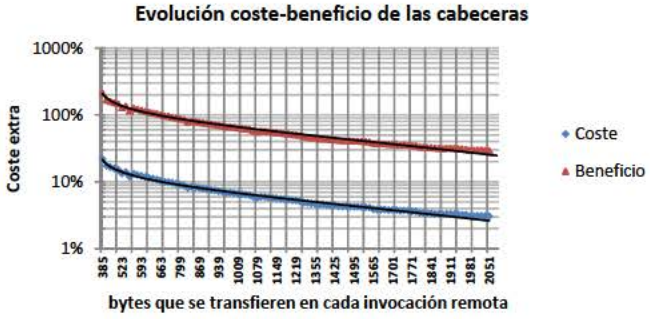

Figura 5. Evolución de la relación coste beneficio de las cabeceras a medida que aumenta el número de bytes transferidos

B. *Evaluación sobre un Marco de Rendimiento AUTOSAR*

La segunda parte de la evaluación compara tanto las ganancias como los beneficios en un marco diseñado para aplicaciones distribuidas. El marco de comparación está basado en AUTOSAR [25] y ha sido utilizado para evaluar aplicaciones Java de tiempo real distribuidas con anterioridad. Describe aplicaciones que tienen frecuencias de operación entre los 83 Hz y los 0,25 kHz.

Para evaluar la sobrecarga computacional se han estudiado tres casos de invocaciones donde el número de invocaciones remotas es i) bajo (1 invocación remota por ciclo), ii) medio (2 invocaciones de tamaño medio por ciclo) y iii) alto (4 invocaciones en total por ciclo con fragmentación de paquetes).

Se ha medido la reducción máxima (Fig. 6) obtenible que para los tamaños y números de mensajes enviados puede potencialmente liberar hasta 1,2 veces el tiempo total de ciclo de la tarea. En este último caso la tarea no sería fácil de utilizar en un régimen continuo pues superaría su período pero si podría ser un coste transitorio asociado a alguna operación de reconfiguración.

Asimismo, si se analiza el coste total introducido por las cabeceras se puede ver cómo se mueve desde tiempo bajos entre un 10% hasta un máximo de un 40 % del tiempo disponible de la aplicación (ya en un situación de sobrecarga) (Fig. 7). Los resultados muestran que el coste extra introducido por las caberas de multiplexación son siempre menores que el beneficio potencialmente introducido (Fig. 8), resultando en beneficio neto para la aplicación.

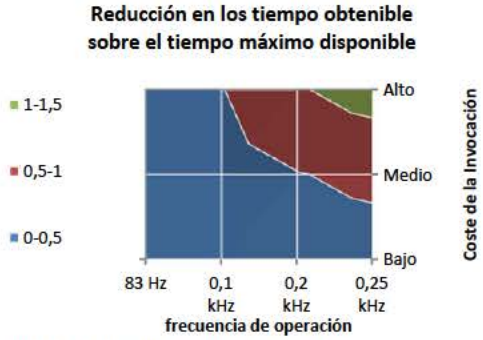

Figura 6. Reducción máxima potencial

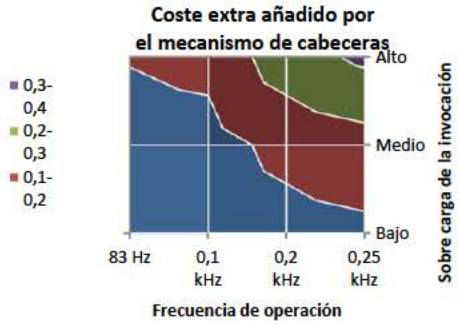

Figura 7. Coste máximo potencial

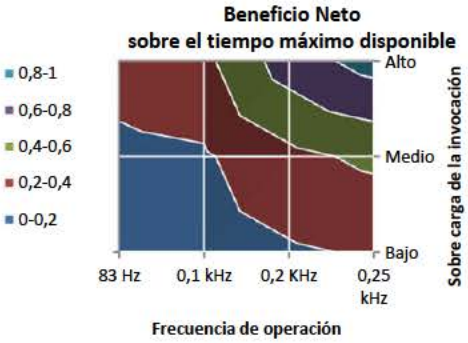

Figura 8. Saldo neto entre el coste y beneficio

V. TRABAJO RELACIONADO

Este trabajo resulta de interés para la comunidad Java de tiempo real. Una de sus líneas de trabajo está relacionada con

la obtención de modelos predecibles para Java de tiempo real distribuido ([26][27][28]). En esta línea, las cabeceras evaluadas servirían para reducir los tiempos de respuesta de los diferentes casos donde hay que dotar al sistema de flexibilidad operacional.

Aunque el modelo es exportable a diferentes tecnologías de comunicación basadas en diferentes mecanismos de comunicación con SOAs ([29][30][31][32, 33]) o RT-CORBA ([34,35]), el diseño del mecanismo está especialmente indicado para RMI y por tanto puede beneficiar a una parte de la comunidad interesada en la utilización de invocaciones remotas basadas en RMI.

La Universidad Politécnica de Madrid (UPM) ha analizado (en [36]) la problemática de Java de tiempo real distribuido y ha producido diferentes marcos de programación para RT-RMI ([15][37]). Ninguno de ellos considera la multiplexación eficiente de diferentes invocaciones basadas en un identificador como el propuesto en este trabajo. Dichos marcos recurren a la creación en una fase inicial de todas las comunicaciones necesarias para comunicarse con el servidor. Por tanto, el trabajo descrito en este artículo les sería útil a la hora de introducir una alternativa que requiriese un menor número de conexiones entre cliente y servidor.

La Universidad de York también ha incluido un marco de trabajo general para introducir Java de tiempo real y distribuido sobre RMI [17]. Su marco general permite reducir la inversión de prioridad de las aplicaciones distribuidas basadas en RMI. En este sentido, las cabeceras propuestas añaden la posibilidad de que se puedan utilizar las comunicaciones subyacentes para establecer comunicaciones sobre un mismo transporte TCP/IP.

Por último, el marco de trabajo denominado DREQUIEMI y propuesto por la Universidad Carlos III de Madrid es uno de los que más se podría beneficiar de este tipo de cabeceras. Los esfuerzos en DREQUIEMI han estado enfocados a la definición de optimizaciones para las invocaciones remotas y a soporte predecible ([23][38][39][24][16, 40][41]); en menor grado se han integrado servicios mejorados y abstracciones de orden superior ([42, 43][44][45, 46]). La integración de las cabeceras podría incluirse directamente sobre el protocolo de tiempo real propuesto en [16].

Por último, este trabajo está fuertemente cimentado en dos trabajos previos donde se proponía un nuevo subprotocolo (denominado *ConnectionLess* [18][19]) para JRMP. Ambos trabajos (el presente y *ConnectionLess*) comparten la idea de tener un mecanismo que permita multiplexación eficiente de comunicaciones. Las diferencias aparecen a la hora de implementar dichas estrategias. *ConnectionLess* propone un nuevo subprotocolo que se añade con el resto ya existente, mientras que el presente trabajo no requiere un nuevo subprotocolo e implementa la extensión directamente sobre uno ya existente. Eso lo hace más sencillo de implementar y también un poco más eficiente pues la plataforma no requiere distinguir entre diferentes tipos de subprotocolos.

VI. CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURO

En este trabajo se ha evaluado la posibilidad de integrar cabeceras de multiplexación directamente sobre el principal subprotocolo de JRMP denominado stream. Para ello, se han añadido dos caberas adicionales sobre los mensajes utilizados durante la invocación remota. Los resultados obtenidos en la evaluación sobre un escenario procedente de un marco de evaluación tipo AUTOSAR muestran unos costes moderados para los casos estudiados y unos grandes beneficios sobre la opción actual existente en JRMP cuando es necesario abrir conexiones paralelas.

A fin de extender el ámbito de aplicación de la estrategia a otros dominios, actualmente se está analizando la aplicación a casos de uso tomados de [47] y [48].

AGRADECIMIENTOS

Este trabajo ha sido financiado parcialmente por el por el proyecto nacional REM4VSS (TIN 2011-28339) y eMadrid (S2013/ICE-2715) y HERMES-SMART-DRIVER(TIN2013-46801-C4-2-R).

REFERENCIAS

[1] J. White, B. Dougherty, R. E. Schantz, D. C. Schmidt, A. A. Porter and A. Corsaro. R&D challenges and solutions for highly complex distributed systems: A middleware perspective. J.Internet Services and Applications 3(1), pp. 5-13. 2012.

[2] D. C. Schmidt, A. Gokhale, R. E. Schantz and J. P. Loyall. Middleware R&D challenges for distributed real-time and embedded systems. *SIGBED Review 1(1),* 2004.

[3] B. Bouyssounouse and J. Sifakis. *Embedded Systems Design The ARTIST Roadmap for Research and Development* 2005.

[4] J. Chen, M. Díaz, L. Llopis, B. Rubio and J. M. Troya. A survey on quality of service support in wireless sensor and actor networks: Requirements and challenges in the context of critical infrastructure protection. *J.Netw.Comput.Appl. 34(4),* pp. 1225-1239. 2011.

[5] V. C. Gungor and G. P. Hancke. Industrial wireless sensor networks: Challenges, design principles, and technical approaches. *Industrial Electronics, IEEE Transactions on 56(10),* pp. 4258. 2009.

[6] E. A. Lee. Cyber physical systems: Design challenges. Presented at International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC). 2008,

[7] W. He and L. D. Xu. Integration of distributed enterprise applications: A survey. *Industrial Informatics, IEEE Transactions PP(99),* pp. 1. 2012.

[8] M. García-Valls, I. Rodríguez-López and L. Fernández-Villar. iLAND: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems. *Industrial Informatics, IEEE Transactions on* pp. -. 2012.

[9] L. D. Xu. Enterprise systems: State-of-the-art and future trends. *Industrial Informatics, IEEE Transactions on 7(4),* pp. 630. 2011.

[10] JSR-50. Distributed real-time specification. [Online]. 2000. Available: http://www.jcp.org/en/jsr/detail?id=50.

[11] J. S. Anderson and E. D. Jensen. Distributed real-time specification for java: A status report (digest). Presented at JTRES '06: Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems. 2006.

[12] Sun Microsystems. Java remote method invocation. 2004. Available: http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf.

[13] J. Waldo, G. Wyant, A. Wollrath and S. C. Kendall. A note on distributed computing. Presented at Mobile Object Systems. 1996.

[14] A. Wollrath, R. Riggs and J. Waldo. A distributed object model for the java system. Presented at COOTS. 1996,

[15] Pablo Daniel Tejera-Carballa, "Communicating middleware for distributed hard real-time systems in java," 2012.

[16] P. Basanta-Val, M. Garcia-Valls and I. Estevez-Ayres, "An architecture for distributed real-time java based on RTSJ and RMI," in *15th IEEE Conference on Emerging Technologies and Factory Communication,* 2010, pp. 1-8.

[17] A. Borg and A. J. Wellings. A real-time RMI framework for the RTSJ. Presented at Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on. 2003.

[18] P. Basanta-Val, M. García-Valls, I. Estévez-Ayres and J. Fernandez-Gonzalez. Integrating multiplexing facilities in the set of JRMP subprotocols. *Latin America Transactions, IEEE (Revista IEEE America Latina) 7(1),* pp. 107-113. 2009. Available: 10.1109/TLA.2009.5173472.

[19] P. Basanta-Val, M. García-Valls, J. Fernandez-Gonzalez and I. Estevez-Ayres, "Fine tuning of the multiplexing facilities of Java's Remote Method Invocation," *Concurrency and Computation Practice and Experience,* vol. 23, pp. 1236-1260, 2011.

[20] P. Basanta-Val "The DREQUIEMI middleware homepage," On line [2015] at http://www.it.uc3m.es/drequiem/drequiemi/, 2015.

[21] E. D. Jensen. The distributed real-time specification for java: An initial proposal. *Comput.Syst.Sci.Eng. 16(2),* pp. 65-70. 2001.

[22] A. J. Wellings, R. Clark, E. D. Jensen and D. Wells. The distributed real-time specification for java: A status report. Presented at Embedded Systems Conference. 2002.

[23] P. Basanta-Val, M. Garcia-Valls and I. Estevez-Ayres. A dual programming model for distributed real-time java. *Industrial Informatics, IEEE Transactions on 7(4),* pp. 750-758. 2011.

[24] P. Basanta-Val, M. Garcia-Valls and I. Estevez-Ayres. Towards propagation of non-functional information in distributed real-time java. Presented at Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on. 2010.

[25] AUTOSAR. Release 4.0 overview and revision history. 2012. Available: www.autosar.org.

[26] P. Basanta-Val and J. S. Anderson, "Using real-time java in distributed systems: Problems and solutions," in *Distributed and Embedded Real-Time Java Systems*, T. H. Toledano and A. J. Wellings, Eds. Springer, 2012, pp. 23-45.

[27] A. J. Wellings, R. Clark, E. D. Jensen and D. Wells. A framework for integrating the real-time specification for java and java's remote method invocation. Presented at Symposium on Object-Oriented Real-Time Distributed Computing. 2002.

[28] E. D. Jensen. A proposed initial approach to distributed real-time java. Presented at ISORC. 2000,

[29] W3C. Web services description language. 2004. Available: http://www.w3.org/TR/2004/WD-wsdl20-primer-20041221/.

[30] M. García-Valls, I. Estévez-Ayres, P. Basanta-Val and C. Delgado-Kloos. CoSeRT: A framework for composing service-based real-time applications. Presented at Business Process Management Workshops 2005. 2005.

[31] I. Estévez-Ayres, M. García-Valls, P. Basanta-Val and J. Díez-Sánchez. A hybrid approach for selecting service-based real-time composition algorithms in heterogeneous environments. *Concurrency and Computation Practice and Experience 23(15),* pp. 1816-1851. 2011.

[32] T. Cucinotta, A. Mancina, G. F. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo and F. Rusina. A real-time service-oriented architecture for industrial automation. *Industrial Informatics, IEEE Transactions on 5(3),* pp. 267. 2009.

[33] T. Cucinotta, L. Palopoli, L. Abeni, D. Faggioli and G. Lipari. On the integration of application level and resource level QoS control for real-time applications. *Industrial Informatics, IEEE Transactions on 6(4),* pp. 479. 2010.

[34] OMG. Real time corba specification version 1.2. 2005. Available: http://www.omg.org.

[35] A. S. Krishna, D. C. Schmidt and R. Klefstad. Enhancing real-time CORBA via real-time java features. Presented at ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04). 2004 .

[36] M. A. d. Miguel. Solutions to make java-RMI time predictable. Presented at Object-Oriented Real-Time Distributed Computing, 2001. ISORC - 2001. Proceedings. Fourth IEEE International Symposium on. 2001 .

[37] D. Tejera, R. Tolosa, M. A. d. Miguel and A. Alonso. Two alternative RMI models for real-time distributed applications. Presented at ISORC '05: Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05). 2005

[38] P. Basanta-Val, M. Garcia-Valls and I. Estevez-Ayres. Non-functional information transmission patterns for distributed real-time java. *Software Practice and Experience 41(12),* pp. 1409-1435. 2011.

[39] P. Basanta-Val, M. García-Valls and I. Estévez-Ayres, "Extending the concurrency model of the Real-Time Specification for Java," *Concurrency and Computation Practice and Experience,* vol. 23, pp. 1623-1645, 2011.

[40] P. Basanta-Val, M. Garcia-Valls and I. Estevez-Ayres, "Using switched-ethernet and linux TC for distributed real-time java infrastructures," in *Work-in-Progress Proceedings IEEE RTAS 2010,* 2010, .

[41] P. Basanta-Val, M. Garcia-Valls and I. Estevez-Ayres. No-heap remote objects for distributed real-time java. *ACM Trans.Embed.Comput.Syst. 10(1),* pp. 1-25. 2010.

[42] P. Basanta-Val and García-Valls M., "Towards a reconfiguration service for distributed real-time java," in *Workshop on Real-Time and Distributed Computing in Emerging Applications (REACTION 2012),* San Juan (Puerto Rico), 2012, .

[43] P. Basanta-Val, M. Garcia-Valls and I. Estevez-Ayres. Enhancing OSGi with real-time java support. *Software Practice and Experience -(-),* pp. ---. 2012.

[44] P. Basanta-Val, M. Garcia-Valls and I. Estevez-Ayres. Towards a cyber-physical architecture for industrial systems via real-time java technology. *Computer and Information Technology, International Conference on 0*pp. 2341-2346. 2010.

[45] P. Basanta-Val, I. Estevez-Ayres, M. Garcia-Valls and L. Almeida. A synchronous scheduling service for distributed real-time java. *Parallel and Distributed Systems, IEEE Transactions 21(4),* pp. 506. 2010.

[46] P. Basanta-Val, L. Almeida, M. Garcia-Valls and I. Estevez-Ayres. Towards a synchronous scheduling service on top of a unicast distributed real-time java. Presented at Real Time and Embedded Technology and Applications Symposium, 2007.RTAS '07.13th IEEE. 2007, .

[47] P. Basanta-Val, N. Fernandez-Gonzalez, A. Wellings, and N. Audsley, "Improving the predictability of distributed Stream Processors". Future Generation Computer Systems 01/2016; 52. DOI:10.1016/j.future.2015.03.023

[48] Feng Lin, Xiangxu Dong, B. M. Chen, Kai-Yew Lum and T. H. Lee. A robust real-time embedded vision system on an unmanned rotorcraft for ground target following. *Industrial Electronics, IEEE Transactions on 59(2),* pp. 1038-1049. 2012.